Software Verification with CPAchecker 3.0: Tutorial and User Guide

Daniel Baier, Dirk Beyer ✉, Po-Chun Chien, Marie-Christine Jakobs,
Marek Jankola, Matthias Kettl, Nian-Ze Lee, Thomas Lemberger,
Marian Lingsch-Rosenfeld, Henrik Wachowitz, and Philipp Wendler

LMU Munich, Munich, Germany

https://cpachecker.sosy-lab.org

Abstract. This tutorial provides an introduction to CPAchecker for users. CPAchecker is a flexible and configurable framework for software verification and testing. The framework provides many abstract domains, such as BDDs, explicit values, intervals, memory graphs, and predicates, and many program-analysis and model-checking algorithms, such as abstract interpretation, bounded model checking, Impact, interpolation-based model checking, *k*-induction, PDR, predicate abstraction, and symbolic execution. This tutorial presents basic use cases for CPAchecker in formal software verification, focusing on its main verification techniques with their strengths and weaknesses. An extended version also shows further use cases of CPAchecker for test-case generation and witness-based result validation. The envisioned readers are assumed to possess a background in automatic formal verification and program analysis, but prior knowledge of CPAchecker is not required. This tutorial and user guide is based on CPAchecker in version 3.0. This user guide's latest version and other documentation are available at https://cpachecker.sosy-lab.org/doc.php.

Keywords: CPAchecker · Configurable Program Analysis · Formal Verification · Model Checking · Software Verification · Program Analysis · Testing · Tutorial · Correctness Certification · Witnesses · Witness Validation · Fault Visualization

1 Introduction

CPAchecker [34] is a framework for configurable software verification with a focus on the verification of C programs. It is based on the concept of configurable program analysis [25, 27, 28] and provides an extensive collection of verification algorithms and abstract domains. Throughout the past years, CPAchecker has been a top contender in the International Competition on Software Verification [11, 12, 13] and has helped identify over 240 bugs in Linux device drivers [43, 63, 83].

An extended version of this user guide is available in a technical report [7].

Fig. 1: Inputs and outputs of CPAchecker when it is used as a verifier, witness validator, or test-case generator

CPAchecker is open source, written in Java, and maintained by an active community (project statistics can be found on OpenHub.net). It puts a high priority on extensibility and flexible reuse of components for developers. The architecture and features of the framework are described in other articles [34, 47]. More information about the achievements, history, and licensing model of CPAchecker are available in the extended version [7].

1.1 Use Cases of CPAchecker

There are three main use cases of CPAchecker, with their inputs and outputs summarized in Fig. 1: (1) As a *verifier*, CPAchecker takes as input a program and a specification, and returns a verdict, a verification report, and a verification witness. The verdict specifies whether the given program adheres to the specification, the verification report allows users to examine the verification result, and the witness contains a machine-readable justification for the returned verdict. (2) As a *witness validator* [5, 18], CPAchecker takes as input a program, a specification, and a witness, and returns a verdict that indicates whether the witness could be confirmed by CPAchecker. (3) As a *test-case generator* [31, 51, 74], CPAchecker takes as input a program and a test-coverage specification, and returns a set of test cases that cover the program according to the specification.

CPAchecker is also used for program transformation [30, 32, 33, 40, 44], to explore decompositions of verification problems [4, 26, 36], and to parallelize verification approaches [22, 36]. This tutorial focuses on using CPAchecker as a verifier. Information about CPAchecker as a witness validator and test-case generator is present in the extended version [7].

1.2 Configurable Program Analysis

CPAchecker uses configurable program analysis (CPA) [25, 27, 28] to compute a program's reachable states. A CPA specifies an abstract domain and a precision used to explore a program's reachable states. The abstract domain defines the representation of a program's state, while the precision defines how precise the abstraction should be. Various CPAs have been implemented in CPAchecker, each tailored to handle specific program features and perform a dedicated analysis. CPAs can also be combined to achieve synergy. Furthermore, precisions can be adjusted dynamically [28], making an analysis coarse but efficient, or precise but resource-consuming. CPAchecker automatically adjusts the precisions via counterexample-guided abstraction refinement (CEGAR) [23, 41, 42, 52] or some carefully-designed procedures [15].

1.3 Documentation and Communication

The `README` and directory `doc/` in the CPAchecker project provide useful information for users and developers. For an overview on the architecture, we recommend the tool paper [34] on CPAchecker and the publications regarding the CPA concept [25, 27, 28]. CPAchecker supports various verification algorithms and techniques. The most important techniques in CPAchecker are explained in separate publications, including data-flow and value analysis [15, 25, 41], SMT-based verification algorithms [21, 37, 38], block-abstraction memoization [22, 23, 24, 82], program transformations [30, 32, 33, 40, 44], cooperative verification [16, 19], witness certification and validation [5, 18], and test-case generation [31, 51, 74]. The configurations of CPAchecker that were submitted to competitions are described in the competition contribution papers of SV-COMP [2, 3, 8, 54, 56, 65, 67, 68, 69, 73, 80, 81], Test-Comp [29, 59, 60, 61], and RERS [45, 46]. These publications give an indication of the breadth of analyses available in CPAchecker and its power and flexibility as a verification framework.

Questions, bug reports, and feature requests for CPAchecker are always welcome on its mailing list (`https://groups.google.com/g/cpachecker-users`) and the issue tracker (`https://gitlab.com/sosy-lab/software/cpachecker/-/issues`).

1.4 CPAchecker in Education

Due to the many algorithms and abstract domains, and the clean and extensible architecture, CPAchecker is an ideal tool for teaching of program-analysis techniques. The techniques can be explored in comparison and their effects observed. Visualizations of abstract states and error paths help understand the reasons for correctness or violation of the specification. We use CPAchecker in various courses on software engineering, software verification, software testing, and program semantics.

1.5 Outline

This tutorial starts in Sect. 2 with installation instructions and a first example of running CPAchecker. Section 3 explains the inputs and outputs of CPAchecker. Finally, Sect. 4 gives an overview on the most important analysis techniques that CPAchecker provides for software verification.

The extended version [7] includes further information on CPAchecker, provides an overview of all concrete example command lines together with references to the respective part of the tutorial, provides more information about the CPAchecker project, its development history, achievements, and licensing, provides some more detailed examples for the presented analysis techniques, explains how to use CPAchecker for witness validation, and explains how to use CPAchecker for test-case generation.

2 Getting Started with CPACHECKER

In the following, we explain the installation and a few alternatives for executing CPAchecker on individual verification tasks.

For trying out CPAchecker and following this tutorial we provide a few example programs in a reproduction package [6]. We assume this package was downloaded and unpacked, and that the current working directory is its root directory (where directory `examples/` is visible). The execution of each example in this tutorial should take less than 10 seconds.

2.1 Local Installation

Installation Requirements. Most features of CPAchecker require a 64-bit GNU/Linux machine, unless users build the required libraries themselves. A limited feature set is usable on other platforms. We recommend a current LTS version of Ubuntu; recent versions of other distributions can be expected to work as well.

Installation. For users on Debian or Ubuntu we provide a package repository at `https://apt.sosy-lab.org`. Please follow the instructions on that webpage to enable the repository. Afterwards, the latest version of CPAchecker can be installed with `sudo apt install cpachecker`.

For users without root access or on other distributions, we also provide CPAchecker as pre-built binary releases via Zenodo [48] and our download page. Please ensure that a Java Runtime Environment (JRE) is available (for CPAchecker 3.0, Java version 17 or newer is required). Unpack the archive for CPAchecker after the download. We recommend adding CPAchecker's `bin/` directory to the `PATH` environment variable. This way the examples provided in this tutorial work as is, without having to specify the full path to the `cpachecker` executable every time. If CPAchecker was installed via the package repository, changing the `PATH` variable is not necessary.

Execution. To try out CPAchecker, run the following command from the reproduction package's [6] root directory:

```
cpachecker examples/example-safe.c
```

This will verify that there is no assertion violation in program `example-safe.c`, and report that the program satisfies the specification. Further information is provided in Sect. 2.4.

2.2 Execution via Container

CPAchecker is available as an image in OCI format, for use with container runtimes like Podman and Docker. The identifiers of the images are `sosylab/cpachecker` (always the latest release) and `sosylab/cpachecker:3.0` for version 3.0. The following command line executes CPAchecker 3.0 from a container (may require `sudo`, depending on the Docker installation):

```
docker run -v "$(pwd)":/workdir sosylab/cpachecker:3.0 \
    examples/example-safe.c
```

Command-line argument `-v "$(pwd)":/workdir` makes the current working directory (`$(pwd)`) available in the started container at path `/workdir`. This is the default entrypoint of the CPAchecker images. Command-line argument `-u $UID:$GID` might be added after `docker run` to set the user and group ID of the container to the current user and group ID: output files produced by CPAchecker are then owned by the current user instead of `root`. Argument `examples/example-safe.c` is passed to CPAchecker and will be explained in Sect. 2.4. The command-line arguments and input files can be adjusted as usual.

2.3 Remote Execution via Website

We provide a web interface for CPAchecker at `https://vcloud.sosy-lab.org/cpachecker/webclient/run/`. The examples of this paper are available as Examples on the left of the page.

2.4 Example Verification Task

For the following example command lines we assume a local installation of CPAchecker and that the artifact with the examples [6] has been unpacked in the current directory (such that the directory `examples/` is present). If necessary, e.g., for Docker usage, please adjust the command lines accordingly.

Program Description. We use the program in Fig. 2a. This program initializes variables `n` and `x` to two nondeterministic but concrete values of type `unsigned int` (modeled by calls to `__VERIFIER_nondet_uint()`) and then initializes `y` to the difference between `n` and `x`. As long as `x` is larger than `y`, the `while` loop decrements `x` and increments `y` by one. If the sum of `x` and `y` does not equal `n` at the end of a loop iteration, `__assert_fail` at line 10 triggers a program error (arguments omitted for simplicity). The program is correct with respect to the specification that `__assert_fail` is unreachable, because the sum of `x` and `y` always equals `n` at the end of every loop iteration. A variant of this program is shown in Fig. 2b. The variant follows the same execution except at line 9. Here an error is triggered if `x` is smaller than `y`. This error is reachable by initializing `n` to 3 and `x` to 2 (among many other possibilities).

Verification Run. To verify the example program in Fig. 2a with CPAchecker, execute the below command in a terminal (cf. example default on the web service):

```
cpachecker examples/example-safe.c
```

This command line does not specify an explicit configuration. In this case CPAchecker uses the default configuration, which is the currently recommended

```c
 1  extern unsigned
        __VERIFIER_nondet_uint();
 2  extern void __assert_fail();
 3  int main() {
 4    unsigned n =
          __VERIFIER_nondet_uint();
 5    unsigned x =
          __VERIFIER_nondet_uint();
 6    unsigned y = n - x;
 7    while(x > y) {
 8      x--; y++;
 9      if (x + y != n) {
10        __assert_fail();
11      }
12    }
13    return 0;
14  }
```

(a) `example-safe.c` (error unreachable)

```c
 1  extern unsigned
        __VERIFIER_nondet_uint();
 2  extern void __assert_fail();
 3  int main() {
 4    unsigned n =
          __VERIFIER_nondet_uint();
 5    unsigned x =
          __VERIFIER_nondet_uint();
 6    unsigned y = n - x;
 7    while(x > y) {
 8      x--; y++;
 9      if (x < y) {
10        __assert_fail();
11      }
12    }
13    return 0;
14  }
```

(b) `example-unsafe.c` (error reachable)

Fig. 2: Example C programs

configuration. Like most configurations shipped with CPAchecker, the default configuration uses the default specification, which specifies that no C assertion error `__assert_fail` and no label named `ERROR` should be reachable. The specifications, configurations, and the available analyses are described in more detail in Sect. 3.2, Sect. 3.3, and Sect. 4.

At the end of its execution, CPAchecker produces the following messages:

```
Verification result: TRUE. No property violation found by chosen configuration.
More details about the verification run can be found in the directory "./output".
Graphical representation included in the file "./output/Report.html".
```

The verification result `TRUE` indicates that the error (line 10 in Fig. 2a) is not reachable. We can also change the input program to `example-unsafe.c` in the command line. In this case, the verification result is `FALSE`, meaning that CPAchecker finds an execution path that triggers the error. The meanings of verification results and how to navigate through the generated report is the topic of Sect. 3.4 and Sect. 3.5, respectively.

3 Input and Output Interface of CPAchecker

Figure 1 shows the inputs and outputs of CPAchecker. CPAchecker always takes a program, a specification, and a configuration as input. It always produces a verdict and a report. Depending on how the user intends to use it, either as a verifier, a witness validator, or a test-case generator, CPAchecker may also take a verification witness as input, or produce witnesses or test cases as output.

3.1 Input Program

CPAchecker supports a large subset of the GNU-C11 features. Normally, the verifier expects pre-processed input files. CPAchecker supports compiler directives (e.g., `#include` or `#define`) if the command-line argument `--preprocess`

Table 1: Provided specifications (files in `config/specification/`)

Specification	Description
`ErrorLabel`	Labels named `ERROR` (case insensitive) are never reachable.
`Assertion`	All `assert` statements hold.
`default`	Both `ErrorLabel` and `Assertion` hold.
`overflow`	All operations with a signed-integer type never produce values outside the range representable by the respective type.
`datarace`	Concurrent accesses to the same memory location must be atomic if at least one of them is a write access.
`memorysafety`	All memory deallocations and pointer dereferences are valid and all allocated memory is pointed to or deallocated when the program exits.
`memorycleanup`	All allocated memory is deallocated before the program exits.

is given, in which case CPAchecker pre-processes the input C program. To guarantee a meaningful verification of programs that use external functions, including functions in the C standard library, the implementations of the functions have to be provided in the input programs. Otherwise, CPAchecker overapproximates their behavior, potentially leading to false alarms. Two exceptions are the function `pthread_create` for creating a new thread and functions `malloc`, `memset`, etc., for manipulating memory, which are handled out-of-the-box by CPAchecker's concurrency and memory analyses, respectively. To verify a software project that consists of multiple C files, all relevant files must be listed on the command-line. By default, CPAchecker starts the analysis from function `main`. Another entry function can be specified with command-line argument `--entry-function <entry function>`.

The semantics of a C program depends on the runtime platform, which consists of a machine architecture, a data model, and an operating system. CPAchecker assumes a single platform during verification. The command-line argument `--32` (default) sets the platform to 32-bit x86 Linux (ILP32) and `--64` sets the platform to 64-bit x86 Linux (LP64) [77].

3.2 Program Specification

Besides the input program, a *specification* is needed as input for CPAchecker. The specification defines what property of the program should be checked. CPAchecker supports an automaton-based specification language (similar to Blast [17] and Slam [9]) to define program specifications (documented in `doc/SpecificationAutomata.md`). CPAchecker ships with several common specifications in the directory `config/specification/`. A selection is listed in Table 1. CPAchecker also supports the property files written in the specification language that was standardized by the International Competition on Software Verification (SV-COMP) [13].

The command-line argument `--spec <specification>` defines the specification to use. It accepts the path to a specification-automaton file, an SV-COMP property file, or the name of one of the specifications that ship with CPAchecker. For

```
1 OBSERVER AUTOMATON AssertionErrorAutomaton
2 INITIAL STATE Init;
3 STATE USEFIRST Init :
4   // AST-based matching of function calls to __assert_fail
5   MATCH {__assert_fail($?)}
6     -> ERROR("assertion in $location");
7 END AUTOMATON
```

Fig. 3: Example of automaton-based specification for checking assert statements

example, to verify a program against the provided specification `Assertion` with CPAchecker's default analysis, we run (cf. example assert on the web service):

```
cpachecker [--preprocess] --spec Assertion examples/example-safe.c
```

The square brackets in the above command indicate that argument `--preprocess` may be omitted if the program does not contain compiler directives (cf. Sect. 3.1).

Figure 3 shows a simplified version of specification `Assertion`. The specification is violated if a call to function `__assert_fail` is reachable in the given input program, which matches how `assert` statements appear in a C program after pre-processing. The automaton starts in the initial state `Init` and observes the analyzed program operations until an operation matches a call to `__assert_fail` (line 5) with an arbitrary number of function-call arguments (denoted by `$?`). In this case, the automaton transitions to the special state `ERROR` (line 6) that signals a specification violation with the given explanation.

3.3 CPAchecker Configuration

CPAchecker is highly configurable via a set of configuration options, which are documented in file `doc/ConfigurationOptions.txt`. Configuration options are specified as key-value pairs in a configuration file or on the command line. An extensive set of bundled configuration files is available in directory `config/`. Most of these bundled configurations specify default values for common configuration options, e.g., the specification `config/specification/default.spc` and a time limit of 900 s. Command-line arguments overwrite these defaults.

It is possible to write and provide own configuration files. Their format is inspired by Windows INI files with some extensions like include directives. A full description is available in `doc/Configuration.md`. Configuration files may use relative paths. CPAchecker interprets these relative paths relative to the directory of the respective configuration file.

Command-line argument `--config CONFIG_FILE` selects a configuration file. The bundled configuration files can also be selected with short-hand arguments that consist of the base name of the configuration file, e.g., `--kInduction` for configuration file `config/kInduction.properties` or `--svcomp24` for configuration file `config/svcomp24.properties`. When no configuration file is explicitly specified, CPAchecker runs in its default configuration (defined by the configuration file `config/default.properties`).

Command-line argument `--option key=value` sets a single configuration option. The order of command-line arguments is irrelevant. If an option is set both in the configuration file and through `--option`, the `--option` value takes precedence and overwrites any value from the configuration file.

CPAchecker provides shortcuts for the most common configuration options, for example `--64` to specify the platform or `--timelimit` to set an analysis time limit. A full list of shortcuts is available via `cpachecker -h` and in `doc/Configuration.md`. For technical reasons, a few command-line arguments exist that can only be specified through command-line arguments and not via configuration files. These arguments include `--benchmark` (which leads to better performance by disabling CPAchecker-internal assertions, writing no output files, and much more) and `--heap` (which adjusts the amount of memory used by the JVM for CPAchecker).

As an example, consider the following command line (cf. example settingOptions on the web service):

```
cpachecker --kInduction --timelimit 900s --heap 2000M \
    --spec ErrorLabel examples/example-safe.c \
    --option solver.solver=MATHSAT5
```

This invokes CPAchecker with the configuration for k-induction, sets the configuration option `limits.time.cpu` for the time limit to 900 s, tells the JVM to use 2 000 MiB of heap memory, chooses the specification file `ErrorLabel`, the program `program.c` as input file, and sets the configuration option `solver.solver` to `MATHSAT5`.

3.4 Verification Verdict

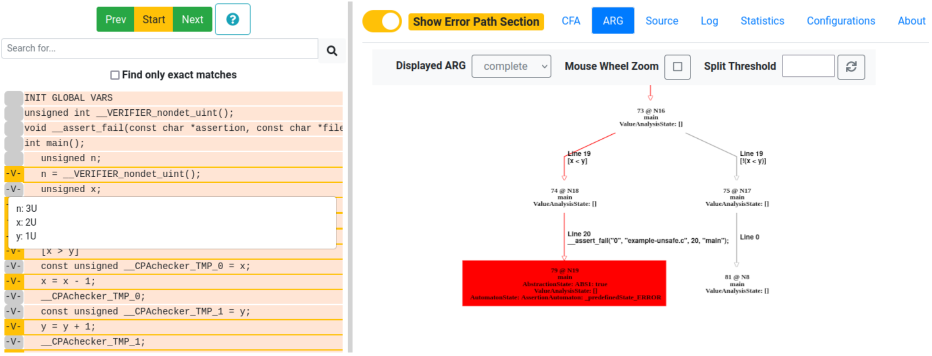
CPAchecker may report three different verification verdicts: (1) `TRUE`, if it proves that the program *satisfies* the specification; (2) `FALSE`, if it proves that the program does *not satisfy* the specification; (3) `UNKNOWN`, if it cannot decide the verification task using the given resource limits and configuration.

3.5 Interactive Report in HTML Format

In addition to a verification verdict, CPAchecker produces detailed information about the performed analysis in directory `output/` in the current working directory. This usually includes an interactive report in HTML format. Note that different configurations may produce different output files.

The interactive report offers a graphical interface for users to inspect the results of CPAchecker. It allows to inspect, among others: the *control-flow automata* (CFA) of the input program, the *abstract reachability graph* (ARG) that was constructed by the chosen configuration, statistics, and an error path that violates the specification (if the verdict is `FALSE`).

In the following we explain the most important parts of this report. A screenshot of the report is shown in Fig. 4. An example report is provided online. When CPAchecker reports verdict `FALSE`, the report file is `output/`



```
    INIT GLOBAL VARS
    unsigned int __VERIFIER_nondet_uint();
    void __assert_fail(const char *assertion, const char *file
    int main();
       unsigned n;
-V-    n = __VERIFIER_nondet_uint();
-V-    unsigned x;
-V-    [x > y]
-V-    const unsigned __CPAchecker_TMP_0 = x;
-V-    x = x - 1;
-V-    __CPAchecker_TMP_0;
-V-    const unsigned __CPAchecker_TMP_1 = y;
-V-    y = y + 1;
-V-    __CPAchecker_TMP_1;
```

```
n: 3U
x: 2U
y: 1U
```

Fig. 4: Screenshot of the HTML report for program `example-unsafe.c`

`Counterexample.0.html` (number `0` may differ). Otherwise, the report file is `output/Report.html`.

Control-Flow Automata. The tab CFA in the report shows the input program in the internal representation of CPAchecker, the control-flow automata (CFA). A CFA consists of program locations (nodes of the graph) and program edges (edges of the graph). In the report, a double-click on a program edge navigates to the source-code line it represents. The drop-down menu "Displayed CFA" can be used to display a single CFA for a single program function.

Abstract-Reachability Graph. The tab ARG in the report shows a graphical representation of the program states that were explored by CPAchecker in the form of an abstract-reachability graph (ARG). The right-hand side of Fig. 4 shows an ARG. Each node in the ARG represents an *abstract* state of the input program. CPAchecker constructs abstract states according to the selected configuration. An abstract state usually represents a set of *concrete* program states in order to overapproximate the reachable state space. Two abstract states are connected by a directed edge if one state is the successor to the other. The directed edge goes from predecessor to successor and is labeled with a program operation that induced the predecessor-successor relation during analysis.

If CPAchecker reported verdict `TRUE`, the ARG represents all reachable abstract program states. If CPAchecker reported verdict `FALSE`, nodes and edges that are part of the error path are marked in red (as in Fig. 4).

Error Path. If the verification verdict is `FALSE`, the report includes a textual error-path section as separate panel on the left (toggle with button "Show Error-Path Section"). This allows users to step through the error path that CPAchecker computed. The textual error path is a list of program statements, accompanied by concrete assignments to all variables on the error path. A button -V- is displayed next to each statement, which indicates the concrete variable assignments at the respective location. To replay the error path step-by-step, users can click on the Start button on the top left. Then, two buttons Next and Prev can be used to navigate through the error path.

```
<...>
content:
- invariant:
    type: "loop_invariant"
    location:
      file_name: "example-safe.c"
      line: 7
      column: 3
      function: "main"
    value: "( x + y == n )"
    format: "c_expression"
```

(a) Relevant sections of correctness witness for safe program in Fig. 2a

```
<...>
content:
 - segment:
   - waypoint:
       type: assumption
       location:
         file_name: "example-unsafe.c"
         line: 6
       constraint:
         value: "x == 0 && n == 1"
 - segment:
   - waypoint:
       type: target
       location:
         file_name: "example-unsafe.c"
         line: 10
```

(b) Relevant sections of violation witness for unsafe program in Fig. 2b

Fig. 5: Example verification witnesses (format version 2.0)

3.6 Statistics

CPAchecker collects a variety of statistics, depending on the chosen analysis. These are presented in the interactive report under tab Statistics and are also written to file `output/Statistics.txt`. With command-line argument `--stats`, CPAchecker prints the statistics to the console at the end of the verification run.

The statistics help users to evaluate the performance of the analysis. Below is an example excerpt of a run's statistics that shows the time spent on SMT solving, the total number of computed reachable abstract states, and the consumed CPU time.

```
Total time for SMT solver (w/o itp):     0.017s
[...]
Size of reached set:             10
[...]
CPU time for analysis:            0.860s
```

A separate tutorial covers how to interpret CPAchecker statistics in more detail.

3.7 Verification Witnesses

Verification witnesses [5, 18] help users and tools to reason about verification results and allow independent validation of the verification result. CPAchecker can both export witnesses for verification results and validate witnesses that other tools produce. The extended version [7] explains witness validation in detail.

Correctness Witnesses. Correctness witnesses are defined for reachability of error locations and detection of signed-integer overflows in sequential programs. CPAchecker produces such a witness not only if the verdict is `TRUE`, but also if it is `UNKNOWN` (in this case with partial information). The witness contains information about the explored program state space in the form of loop and location invariants. In case the analysis result is `TRUE`, the invariants hold whenever the program execution passes through the respective location.

```
 1 unsigned __VERIFIER_nondet_uint() {
 2   static unsigned call_count = 0;
 3   unsigned retval;
 4   switch (call_count) {
 5     case 0: retval = 2U; break;
 6     case 1: retval = 2U; break;
 7   }
 8   ++call_count;
 9   return retval;
10 }
```

Fig. 6: Test harness generated for the example program in Fig. 2b

Figure 5a shows an excerpt of a correctness witness for the safe program in Fig. 2a. It reports the loop invariant `x + y == n` for the loop head in line 7. This helps to prove the program correct.

Violation Witnesses. Violation witnesses represent one or more program paths that lead to a specification violation. This is achieved by specifying assumptions about the program inputs and the control flow of the program.

Figure 5b shows an excerpt of a violation witness for the unsafe program in Fig. 2b. It shows the program path that leads to the assertion failure at line 10 when `x` is assigned value 0 and `n` is assigned value 1.

3.8 Test Harnesses

If CPAchecker finds a specification violation (verdict `FALSE`), it produces a test harness that triggers this violation through test execution. A test harness contains a sequence of external inputs (e.g., for inputs modeled by `__VERIFIER_nondet*`) to the program that trigger an execution path to the specification violation. Figure 6 shows an excerpt of a test harness for the example program in Fig. 2b. The two return values `2U` (lines 5 and 6) initialize, in the program under analysis (Fig. 2b), both variables `n` and `x` with value 2. This triggers the assertion failure at line 10 of the program.

The test harness can be compiled with the program under analysis:

```
gcc output/Counterexample.1.harness.c examples/example-unsafe.c
```

This produces a binary `a.out`. The execution of `./a.out` exhibits that the claimed specification violation is actually reachable. It reports:

```
CPAchecker test harness: property violation reached
```

The extended version [7] gives more details on test generation with CPAchecker.

4 Verification Analyses and How to Select Them

This section shows how to execute various commonly-used verification analyses in CPAchecker. These analyses can be divided into three groups depending on

Table 2: Commonly-used configurations and supported specifications

Configuration	Specification (cf. Sect. 3.2)	Description
Configurations for reachability specifications:		

`--valueAnalysis-NoCegar-join`
`--symbolicExecution-NoCegar`
`--predicateAnalysis`
`--bmc-incremental`
`--kInduction`
	`default`, `Assertion`, `ErrorLabel`, custom automaton specifications, and SV-COMP property `unreach-call.prp`	Section 4.2
		Section 4.4
		Section 4.5
		Section 4.6
		Section 4.7

Special-purpose configurations:		
`--smg`	memory safety (`memorysafety` and `memorycleanup`)	Section 4.8
`--lassoRankerAnalysis` `--terminationToSafety`	termination	Section 4.9
`--predicateAnalysis--overflow`	`overflow`	Section 4.10
`--dataRaceAnalysis`	`datarace`	Section 4
Meta configurations:		
`--svcomp24` default (no argument)	reachability specifications and all SV-COMP properties	[8] Section 4.1

the kind of specifications they can check. First, there are analyses that perform a reachability analysis. These support common specifications, for example, reachability of an error location or an assertion violation. Second, there are analyses that support a particular special-purpose specification. Third, there are meta analyses that implement strategy selection and delegate to one of the above depending on the provided specification. Table 2 lists common configurations and the respective specifications they support. Apart from the configuration `--dataRaceAnalysis`, which performs partial order reduction [72] over memory accesses in combination with value analysis [41], the following sections explain these configurations in more detail.

4.1 Selecting an Analysis

Selecting an analysis of CPAchecker primarily depends on the kind of specification that should be verified. Memory safety, overflows, and data races can each be verified by exactly one recommended analysis, which is listed in Table 2. For termination, there are two recommendations, described in Sect. 4.9. If SV-COMP property files are used to encode the specification, meta configurations of CPAchecker automatically select a recommended analysis depending on the specification.

For standard reachability specifications a wide range of different analyses and techniques is available in CPAchecker. Each of them has their strengths and weaknesses, and while some of them are more powerful or efficient in general, none of them always outperforms all of the others, so it can be worthwhile experimenting with several analyses.

The general recommendation for most use cases is the default analysis of CPAchecker (used if no other configuration is selected on the command line).

Table 3: Main configuration flavors of value analysis

Precision Refinement	Path Sensitivity	Configuration
✗	✗	`--valueAnalysis-NoCegar-join`
✗	✓	`--valueAnalysis-NoCegar`
✓	✓	`--valueAnalysis-Cegar`

It is a meta configuration that uses k-induction (`--kInduction`, most effective overall in our experience) for reachability specifications.

CPAchecker's value analysis (`--valueAnalysis-NoCegar-join`), symbolic execution (`--symbolicExecution-NoCegar`), and bounded model checking (BMC, `--bmc-incremental`) are mostly suited for finding specification violations. While they are often quite efficient in finding bugs, they are often inefficient for proving correctness for large programs. In our experience these configurations usually either succeed quickly or will not produce a result at all.

To prove the absence of specification violations in larger programs, either abstraction of the program state space or a proof technique such as induction needs to be used. Value analysis and symbolic execution support a limited form of abstraction (ignoring irrelevant program variables and clauses) if their configuration variants with precision refinement are chosen as described in the respective sections below. Predicate abstraction (`--predicateAnalysis`) is stronger and can in principle find arbitrary loop invariants as long as the loop invariants do not require quantifiers nor floating-point arithmetic. k-Induction (`--kInduction`) on the other hand requires that an induction proof can be found for the program.

Another aspect that needs to be considered is that value analysis and symbolic execution in CPAchecker do not support precise reasoning about dynamically allocated memory and data structures on the heap, whereas BMC, predicate abstraction, and k-induction do support this. However, the latter three are based on solving (sometimes large) formulas with an SMT solver, which may not scale. Value analysis has the advantage that it does not require SMT solving, but the disadvantage that it cannot reason about non-deterministic values. Symbolic execution uses an SMT solver, but only when required for non-deterministic values.

The value analysis can be considered comparatively easy to understand conceptually, which makes it a good starting point for the use of CPAchecker.

4.2 Value Analysis

CPAchecker's value analysis tracks concrete value assignments. There are two main configuration choices for the value analysis: (1) whether to use precision refinement, and (2) whether to be path sensitive. Table 3 lists the available command-line arguments to run CPAchecker with the corresponding configuration of value analysis. For example, the following command runs a configuration of value analysis that implements constant propagation [1] (no precision refinement, no path sensitivity) on the program in Fig. 7 (cf. example valueAnalysis-NoCegar-join on the web service):

```c
extern void __assert_fail();
int main() {
  int x = 0;
  int y = 0;
  int z = 0;
  while (x < 2) {
    x++;
    y = z + 1;
  }
  if (z != 0) {
    __assert_fail();
  }
  return 0;
}
```

Fig. 7: Program `example-const.c`

```c
extern unsigned __VERIFIER_nondet_uint();
extern void __assert_fail();
int main() {
  unsigned int x = __VERIFIER_nondet_uint();
  unsigned int y = x;
  unsigned int z = __VERIFIER_nondet_uint();
  while (x < 2) {
    x++;
    y++;
    z = x + z;
  }
  if (x != y) {
    __assert_fail();
  }
  return 0;
}
```

Fig. 8: Program `example-sym.c`

```
cpachecker --valueAnalysis-NoCegar-join examples/example-const.c
```

This configuration tracks only value assignments that always hold on a given location, because program paths are joined when their control flow meets. This is efficient, but in most cases not powerful enough to verify programs. For Fig. 7, it suffices because only the value of variable `z` is needed to prove the program safe, and this is always 0. The extended version [7] shows the state-space exploration of the value analysis for this example in more detail. If, however, the program safety would also depend on the values of `x` or `y` after the loop, the verification result would be `UNKNOWN` because the analysis does not track these non-constant variable values.

The value analysis with path sensitivity tracks value assignments per program path and location. For the example in Fig. 7, it would keep track of all variable values and fully unroll the loop. This leads to path explosion when many paths with distinct value assignments exist, because the analysis tracks all of them separately.

Value analysis with path sensitivity and precision refinement mitigates this path explosion by tracking only those value assignments that are necessary for the analysis to prove the program safe. This is more efficient than value analysis without precision refinement in the common case where not all variables in the program are relevant for safety, like in Fig. 7. The relevant variables are detected automatically through counterexample-guided abstraction refinement (CEGAR) with Craig interpolation [41].

Because the value analysis always tracks concrete value assignments and overapproximates nondeterministic values, it may find false alarms. To mitigate this, CPAchecker runs a precise, SMT-based feasibility check on every found potential error path and only reports confirmed specification violations. This can be seen in the output of CPAchecker, which is provided in the extended version [7].

4.3 Interval-Based Data-Flow Analysis

The data-flow analysis (DF) of CPAchecker is a lightweight proof-finding technique that uses *arithmetic expressions over intervals* as its abstract domain [15, 20]. It tracks, for an automatically-selected set of program variables, the range of values that each variable can take in the form of interval expressions, e.g., $[l_1, u_1] \cup [l_2, u_2]$, where l_i (resp. u_i) is a numerical value representing the lower (resp. upper) bound of an interval. DF supports dynamic precision refinement. At the beginning of the analysis, it performs a coarse but efficient program exploration. If some abstract state reachable in the exploration violates the safety specification, DF incrementally increases its precision by tracking more program variables, allowing more complex expressions of intervals, and disabling widening [53]. To run DF in CPAchecker, provide the configuration `--dataFlowAnalysis` on the command line (cf. example dataFlowAnalysis on the web service):

```
cpachecker --dataFlowAnalysis examples/example-const.c
```

For the above example, CPAchecker produces verdict `TRUE`. A limitation of DF is that its abstract program exploration cannot identify concrete error paths when there are specification violations and may sometimes be too imprecise to find a safety proof. For example, when CPAchecker analyzes `example-safe.c` or `example-unsafe.c` in Fig. 2 with DF, it produces verdict `UNKNOWN`. DF cannot only run standalone but also serve as an auxiliary invariant generator that assists other analyses, e.g., k-induction [19] (cf. Sect. 4.7).

4.4 Symbolic Execution

The symbolic execution [39] of CPAchecker tracks concrete value assignments the same way as the value analysis. But for every value that cannot be tracked concretely, for example because it is assigned non-deterministically, symbolic execution introduces a new symbolic value s_i. Whenever a symbolic value is used in an expression, symbolic execution stores the expression over this symbolic value without evaluating it. In addition, symbolic execution tracks the constraints over these symbolic values for each program path. This produces a symbolic-execution tree (cf. the extended version [7] for details). From this, concrete variable assignments can be derived for any program path. The symbolic execution of CPAchecker also supports precision refinement through CEGAR with Craig interpolation [38]. This derives both variable assignments and constraints that must be tracked through the program.

The below command runs a configuration of symbolic execution [64] without precision refinement (cf. example symbolicExecution-NoCegar on the web service):

```
cpachecker --symbolicExecution-NoCegar examples/example-sym.c
```

Because symbolic execution tracks the expressions over symbolic values without further abstraction, it is well suited for collecting constraints on inputs for certain program paths. But this precision also leads to path explosion: The analysis of symbolic execution on program `example-safe.c` (Fig. 2a) does not terminate. To prove the program safe, it is important to know that the sum of x and y equals n at line 9. Symbolic execution tracks this by storing the expressions $n = s_1, x = s_2, y = s_1 - s_2$, $x = s_2 - 1, y = s_1 - s_2 + 1$, $x = s_2 - 1 - 1, y = s_1 - s_2 + 1 + 1$, and so on. This produces ever more complicated expressions and does not scale.

The following command runs a configuration of symbolic execution [64] with precision refinement (cf. example symbolicExecution-Cegar on the web service):

```
cpachecker --symbolicExecution-Cegar examples/example-sym.c
```

On the program of Fig. 8, this only tracks assignments and constraints over x and y that are necessary to prove the program safe. Assignments to z are not tracked.

4.5 Predicate Abstraction

Predicate abstraction [35, 58, 62] abstracts the program's state space with predicates that it learns through CEGAR [52]. Compared to symbolic execution, predicate abstraction is not limited to tracking (symbolic) values in the program, but can derive more powerful abstractions. The computation of abstractions can be costly, thus predicate abstraction uses *adjustable block encoding* [35] to compute abstractions only at certain program locations, which by default are the loop-head locations. This reduces the number of abstractions calculated and, hence, the overall cost. To run predicate abstraction, use the command (cf. example predicateAnalysis on the web service):

```
cpachecker --predicateAnalysis examples/example-safe.c
```

In this example, predicate abstraction derives the loop invariant `x + y == n` that proves that `__assert_fail` in Fig. 2a is unreachable, and hence returns the verdict `TRUE`. Learned predicates at these locations are written down in a format based on SMT-LIB2 [10] into the file `output/predmap.txt` of the current working directory. Take the program in Fig. 2a for example. Predicate abstraction can derive the invariant `x + y == n` for the `while` loop at line 7 in function `main` that suffices to prove the safety specification that the assertion error is unreachable. In `predmap.txt`, this is represented as follows:

```
(declare-fun |main::n| () (_ BitVec 32))
(declare-fun |main::y| () (_ BitVec 32))
(declare-fun |main::x| () (_ BitVec 32))

main:
(assert (= |main::n| (bvadd |main::y| |main::x|)))
```

Predicate abstraction can abstract the program state space very concisely in a way that proves the program safe, if it learns the right predicates. Unfortunately, there is no mechanism forcing predicate abstraction to find predicates that abstract well. Especially for concrete value assignments in the program, the learned predicates might enumerate all possible states. For instance, predicate abstraction may unnecessarily learn the predicates `x == 0`, `x == 1`, and `x == 2` at line 6 of Fig. 7, instead of `z == 0`. Alternatively, Impact [71] is another analysis that abstracts a program's state space with predicates. It computes and refines abstractions in a lazier way compared to predicate abstraction, and can be initiated using the configuration `--predicateAnalysis-ImpactRefiner-ABEl`. The two analyses have shown different and complementing strengths in our empirical evaluations [21]: Predicate abstraction is more effective at deriving proofs, whereas Impact is more efficient at finding specification violations.

4.6 Bounded Model Checking

Bounded model checking (BMC) is an analysis specialized in finding specification violations [21, 50]. Given a bound n, BMC symbolically unrolls the loops in the program n times, encodes all possible execution paths (within the unrolling bound n) violating the safety specification into an SMT formula, and checks the satisfiability of the formula with an SMT solver. The satisfiability of the formula directly corresponds to the feasibility of the encoded paths. If the formula is satisfiable, a specification-violating execution path exists and can be extracted from the satisfying assignment. CPAchecker then reports the verification verdict `FALSE`. In case the formula is unsatisfiable, the program is considered safe up to the bound n. CPAchecker reports the verification verdict `TRUE` if the loops in the program have finite bounds and are fully unrolled by the bound n. Otherwise, the verdict is `UNKNOWN`, as the behavior of the program at higher unrolling bounds is still unknown.

To execute BMC with an unrolling bound n in CPAchecker, use the configuration `--bmc` and set the option `cpa.loopbound.maxLoopIterations` to $n+1$ on the command line. For instance, the following command runs BMC with an unrolling bound of 2 on the example program in Fig. 2b (cf. example bmc on the web service):

```
cpachecker --bmc --option cpa.loopbound.maxLoopIterations=3 \
    examples/example-unsafe.c
```

CPAchecker can also automatically determine the required unrolling bound. The configuration `--bmc-incremental` enables *incremental* BMC. In this case, it is not necessary to specify option `cpa.loopbound.maxLoopIterations`. Incremental BMC performs BMC iteratively. It starts with an unrolling bound of 0 and increments the bound by 1 at each iteration. The analysis terminates when an error path is found, the safety specification is proven (by fully unrolling all loops in the program), or a resource limit is reached. For instance, if CPAchecker verifies the program in Fig. 2a with `--bmc-incremental`, it produces log messages that show the current unrolling bound:

```
Adjusting maxLoopIterations to 2
↪  (LoopBoundCPA:LoopBoundPrecisionAdjustment.nextState, INFO)
```

CPAchecker eventually reaches the time limit and the verdict is `UNKNOWN`, since a really large unrolling bound (roughly 2^{31}) is required to fully explore the program. If the loop condition at line 7 changes to `x > 0 && x < 3` in Fig. 2a, incremental BMC can prove the program safe with 2 loop unrollings.

4.7 Extensions of BMC for Unbounded Verification

BMC can be extended for unbounded verification of programs by employing the k-induction principle [19, 76] or constructing fixed points, i.e., inductive invariants, via Craig interpolation [37, 70, 78, 79]. To run k-induction in CPAchecker, use the configuration `--kInduction`, which combines k-induction with an auxiliary invariant generator based on data-flow analysis [15, 19] (described in Sect. 4.3). The

```c
#include <stdlib.h>
#include <assert.h>
extern int __VERIFIER_nondet_int();
int main() {
  int size = 100;
  int num = __VERIFIER_nondet_int();
  int * arr = malloc(sizeof(int) * size);
  for (int i = 0; i < size; i++) {
    arr[i] = num;
    num++;
  }
  for (int i = size; i >= 0; i--) {
    assert(*(arr + i) == num);
    num--;
  }
  return 0;
}
```

Fig. 9: `example-unsafe-memsafety.c` with two distinct memory-safety violations

invariants produced by the latter are used to strengthen the induction hypotheses of the former. This is more effective than plain k-induction [19]. As opposed to incremental BMC, k-induction could easily prove the safety of the example programs in Fig. 2a with the command (cf. example kInduction on the web service):

```
cpachecker --kInduction examples/example-safe.c
```

CPAchecker has three verification algorithms based on BMC and Craig interpolation: interpolation-based model checking (IMC) [37, 70], interpolation-sequence-based model checking (ISMC) [14, 78], and dual approximated reachability (DAR) [14, 79]. From unsatisfiable BMC queries, the three algorithms derive interpolants to construct inductive invariants at loop heads. Such an invariant overapproximates the reachable states of the program that conforms to the safety specification, and hence could serve as a proof for the program's correctness. IMC, ISMC, and DAR are enabled via the configurations `--bmc-interpolation`, `--bmc-interpolationSequence`, and `--bmc-interpolationDualSequence`, respectively, and currently support only programs with at most one loop. When CPAchecker verifies the program in Fig. 2a with `--bmc-interpolation` via the command (cf. example bmc-interpolation on the web service):

```
cpachecker --bmc-interpolation examples/example-safe.c
```

it produces the below log message:

```
The current image reaches a fixed point
↪  (IMCAlgorithm.reachFixedPointByInterpolation, INFO)
```

The message indicates that IMC has found an inductive invariant for the `while` loop at line 7 and proved the safety specification of the program.

4.8 Symbolic Memory Graphs with Symbolic Execution

CPAchecker's symbolic-memory-graph (SMG) analysis [55] combines symbolic execution [64] with a graph-based domain that tracks all memory. It is usable in

```c
extern unsigned
       __VERIFIER_nondet_uint();
int main() {
  unsigned n = 1;
  unsigned z =
         __VERIFIER_nondet_uint();
  while (n <= z) {
    n = n + 1;
    z = z - 1;
  }
  return 0;
}
```

(a) `example-safe-termination.c`

```c
extern unsigned
       __VERIFIER_nondet_uint();
int main() {
  unsigned n = 1;
  unsigned z =
         __VERIFIER_nondet_uint();
  while (n <= z) {
    n = (n - 1) % 3;
    z = (z + 1) % 3;
  }
  return 0;
}
```

(b) `example-unsafe-termination.c`

Fig. 10: Example C programs for demonstration of termination analyses

CPAchecker with the configuration `--smg`. In addition to common state-space exploration, the SMG analysis can check for memory safety. The analysis can detect memory leaks, invalid memory access, and invalid freeing of memory.

SMGs accurately track most memory operations, including pointer arithmetics and bit-precise reading of memory. They also store memory boundaries and can thus be used to reason about the validity of pointer dereferences. A distinguishing feature of SMGs is that linked lists of arbitrary length can be abstracted under certain circumstances. This is currently limited to lists that terminate in indefinitely repeating equal values. When the analysis fails to abstract lists of arbitrary length, it enumerates all possible list lengths. This may lead to a path explosion, but can still find violations to safety specification.

We can see some capabilities of the SMG analysis on the example program in Fig. 9. The program first allocates some memory at line 7, then uses this memory to store some distinct but non-deterministic values in a loop at line 9, filling the entire memory allocated in `arr`. Then, in a reversed loop, the saved values are compared to their expected values at line 13. Please note that this example is not pre-processed and thus the command-line argument `--preprocess` is needed. To start the verification of memory safety with the configuration `--smg` on this program, run the following command:

```
cpachecker --preprocess --smg --spec memorysafety \
    examples/example-unsafe-memsafety.c
```

This detects that the first memory access of the second loop at line 12 is unsafe (i.e., the verdict is `FALSE`), as the pointer dereference exceeds the bounds of the allocated memory. Another error can be found before line 16, as the memory allocated in `arr` is never freed. This second memory-safety violation can be found either by fixing the invalid dereference at line 13, or by using the dedicated specification `memorycleanup`:

```
cpachecker --preprocess --smg --spec memorycleanup \
    examples/example-unsafe-memsafety.c
```

4.9 Termination Analysis

Specification *termination* requires a program to always terminate. A program that can execute infinitely is called *non-terminating*.

CPAchecker provides two approaches for termination analysis: the liveness-as-safety analysis [75] `--terminationToSafety` and the lasso-based analysis [57] `--lassoRankerAnalysis`. Analysis `--terminationToSafety` is based on loop unrolling (similar to BMC, cf. Sect. 4.6). It can prove termination only if all loops in the program can be fully unrolled, but is often efficient in finding specification violations, i.e., counterexamples that show non-termination. Analysis `--lassoRankerAnalysis` constructs ranking functions and does not need to unroll all loops in the program for termination proofs.

Liveness-as-Safety Analysis. The liveness-as-safety analysis transforms a liveness specification (termination) into a safety one. It stores the information about values of variables that were seen at the programs' loop heads. For example, the loop head for the two programs in Fig. 10 is the location that corresponds to the line with `while`. Similar to BMC (cf. Sect. 4.6), when the analysis visits a loop head for the n-th time, it constructs a SMT formula that symbolically represents the n-th unrolled loop. Via satisfiability queries, the analysis checks whether there exists a reachable state that is visited twice within n loop iterations. If such a state is found, the program is non-terminating.

The following command line runs the analysis on the program in Fig. 10b (cf. example terminationToSafety on the web service):

```
cpachecker --terminationToSafety \
    examples/example-unsafe-termination.c
```

CPAchecker reports verdict `FALSE` and produces a counterexample that shows the following three unrollings of the loop (visible in the output file `output/Counterexample.1.core.txt`):

$$\texttt{(n, z):}\ \texttt{(1, 2)} \to \underline{\texttt{(0, 0)}} \to \texttt{(1, 1)} \to \texttt{(2, 2)} \to \underline{\texttt{(0, 0)}}$$

The unrolling represents an execution with assignment `z = 2` at line 4. By inspecting the values of `n` and `z` at the loop-head location of each iteration, we see that the state `(n,z) = (0,0)` is visited twice. This represents a non-terminating loop.

Lasso-Based Analysis. The main idea of the lasso-based analysis is to extract potentially non-terminating structures called *lassos* and then pass each of them to the library LassoRanker [66]. This library constructs ranking functions, which are arguments for termination. Simultaneously, it is looking for a non-termination argument. If it finds a non-termination argument for at least one lasso, CPAchecker claims the program non-terminating.

The lasso-based analysis complements the liveness-as-safety analysis. The analysis can verify the terminating program `example-unsafe-termination.c` in Fig. 10a, but not the non-terminating program `example-safe-termination.c` in Fig. 10b. The following command line runs it on the program in Fig. 10a (cf. example lassoRankerAnalysis on the web service):

```
cpachecker --lassoRankerAnalysis \
    examples/example-safe-termination.c
```

CPAchecker reports verdict `TRUE` and produces the output file `output/terminationAnalysisResult.txt`. This contains a termination argument in the form of ranking function $3 * z - 3 * n + 4$. As n is always positive, if the loop condition $n \leq z$ is satisfied, $3 * z - 3 * n + 4 \geq 0$ holds. In addition, after each loop iteration, the resulting value of the ranking function strictly decreases. After a finite number of iterations, the value will eventually become smaller than zero,which implies the negation of the loop condition and thus termination.

4.10 Integer-Overflow Detection

To detect integer overflows, CPAchecker uses a standard reachability analysis, such as those explained in Sects. 4.2, 4.5, and 4.6, together with an internal encoding of overflow conditions as error locations (CPAchecker's overflow analysis also checks for underflows). The configurations supporting overflow detection have the suffix `--overflow` in their names. By default, CPAchecker only checks for signed integer overflows, as these are declared undefined behavior by the C standard. To additionally check for unsigned integer overflows, set the option `overflow.checkUnsigned` to `true`. For instance, to determine whether the example program in Fig. 2a is free of signed and unsigned integer overflows while using predicate abstraction (cf. Sect. 4.5), run the command (cf. example predicateAnalysis-unsigned-overflow on the web service):

```
cpachecker --predicateAnalysis--overflow \
    --option overflow.checkUnsigned=true examples/example-safe.c
```

The verification verdict is `FALSE`, because an overflow could happen at line 6 if `n` and `x` are initialized to 0 and 1, respectively.

5 Conclusion

This tutorial gives an introduction to the CPAchecker framework and how to use it to verify programs. It gives an overview of the main analysis techniques that CPAchecker offers, together with their strengths and weaknesses, and provides guidance on how to use CPAchecker in several analysis situations.

We hope that our tutorial is useful for researchers, practitioners, and educators, and that we stimulate interest and curiosity to dig deeper into the full potential of software model checking. Interested readers can find more information on the CPAchecker project web page, in the research publications on CPAchecker, the CPAchecker GitLab repository, and the CPAchecker mailing list.

Data-Availability Statement. CPAchecker is available at its project website `https://cpachecker.sosy-lab.org` and Zenodo [48]. This tutorial uses version 3.0 [49]. We also provide a reproduction package [6] that includes all the examples from this tutorial.

Funding Statement. CPAchecker was funded in part by the Canadian Natural Sciences and Engineering Research Council (NSERC) — 482301, by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY), 418257054 (Coop), 496588242 (IdeFix), 496852682 (ReVeriX), by the Free State of Bavaria, and the LMU PostDoc Support Funds.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In: Proc. TACAS. pp. 355–359. LNCS 10206, Springer (2017). `https://doi.org/10.1007/978-3-662-54580-5_22`
3. Andrianov, P., Mutilin, V., Khoroshilov, A.: CPALockator: Thread-modular approach with projections (competition contribution). In: Proc. TACAS (2). pp. 423–427. LNCS 12652, Springer (2021). `https://doi.org/10.1007/978-3-030-72013-1_25`
4. Apel, S., Beyer, D., Mordan, V.O., Mutilin, V.S., Stahlbauer, A.: On-the-fly decomposition of specifications in software model checking. In: Proc. FSE. pp. 349–361. ACM (2016). `https://doi.org/10.1145/2950290.2950349`
5. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. Springer (2024)
6. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Reproduction package for FM 2024 article 'Software verification with CPAchecker 3.0: Tutorial and user guide. Zenodo (2024). `https://doi.org/10.5281/zenodo.12666378`
7. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPAchecker 3.0: Tutorial and user guide (extended version). arXiv/CoRR (2024)
8. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). `https://doi.org/10.1007/978-3-031-57256-2_21`
9. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, Microsoft Research (2002). `https://www.microsoft.com/en-us/research/publication/slic-a-specification-language-for-interface-checking-of-c/`
10. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., University of Iowa (2010). `https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf`
11. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). `https://doi.org/10.1007/978-3-030-99527-0_20`

12. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
13. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
14. Beyer, D., Chien, P.C., Jankola, M., Lee, N.Z.: A transferability study of interpolation-based hardware model checking for software verification. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3660797
15. Beyer, D., Chien, P.C., Lee, N.Z.: CPA-DF: A tool for configurable interval analysis to boost program verification. In: Proc. ASE. pp. 2050–2053. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00213
16. Beyer, D., Chien, P.C., Lee, N.Z.: Augmenting interpolation-based model checking with auxiliary invariants. In: Proc. SPIN. Springer (2024)
17. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST query language for software verification. In: Proc. SAS. pp. 2–18. LNCS 3148, Springer (2004). https://doi.org/10.1007/978-3-540-27864-1_2
18. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
19. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
20. Beyer, D., Dangl, M., Wendler, P.: Combining k-induction with continuously-refined invariants. Tech. Rep. MIP-1503, University of Passau (January 2015). https://doi.org/10.48550/arXiv.1502.00096
21. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
22. Beyer, D., Friedberger, K.: Domain-independent multi-threaded software model checking. In: Proc. ASE. pp. 634–644. ACM (2018). https://doi.org/10.1145/3238147.3238195
23. Beyer, D., Friedberger, K.: In-place vs. copy-on-write CEGAR refinement for block summarization with caching. In: Proc. ISoLA. pp. 197–215. LNCS 11245, Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_14
24. Beyer, D., Friedberger, K.: Domain-independent interprocedural program analysis using block-abstraction memoization. In: Proc. ESEC/FSE. pp. 50–62. ACM (2020). https://doi.org/10.1145/3368089.3409718
25. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
26. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: Proc. ICSE. pp. 536–548. ACM (2022). https://doi.org/10.1145/3510003.3510064
27. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
28. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE. pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13

29. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
30. Beyer, D., Jakobs, M.C.: Fred: Conditional model checking via reducers and folders. In: Proc. SEFM. pp. 113–132. LNCS 12310, Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_7
31. Beyer, D., Jakobs, M.C.: Cooperative verifier-based testing with CoVeriTest. Int. J. Softw. Tools Technol. Transfer **23**(3), 313–333 (2021). https://doi.org/10.1007/s10009-020-00587-8
32. Beyer, D., Jakobs, M.C., Lemberger, T.: Difference verification with conditions. In: Proc. SEFM. pp. 133–154. LNCS 12310, Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_8
33. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
34. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
35. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010). https://dl.acm.org/doi/10.5555/1998496.1998532
36. Beyer, D., Kettl, M., Lemberger, T.: Decomposing software verification using distributed summary synthesis. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3660766
37. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning (2024), accepted, preprint available via https://doi.org/10.48550/arXiv.2208.05046
38. Beyer, D., Lemberger, T.: Symbolic execution with CEGAR. In: Proc. ISoLA. pp. 195–211. LNCS 9952, Springer (2016). https://doi.org/10.1007/978-3-319-47166-2_14
39. Beyer, D., Lemberger, T.: CPA-SymExec: Efficient symbolic execution in CPAchecker. In: Proc. ASE. pp. 900–903. ACM (2018). https://doi.org/10.1145/3238147.3240478
40. Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M.: CEGAR-PT: A tool for abstraction by program transformation. In: Proc. ASE. pp. 2078–2081. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00215
41. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11
42. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Proc. SPIN. pp. 20–38. LNCS 9232, Springer (2015). https://doi.org/10.1007/978-3-319-23404-5_3
43. Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proc. ISoLA. pp. 1–6. LNCS 7610, Springer (2012). https://doi.org/10.1007/978-3-642-34032-1_1
44. Beyer, D., Rosenfeld, M.L., Spiessl, M.: A unifying approach for control-flow-based loop abstraction. In: Proc. SEFM. pp. 3–19. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_1
45. Beyer, D., Stahlbauer, A.: BDD-based software model checking with CPAchecker. In: Proc. MEMICS. pp. 1–11. LNCS 7721, Springer (2013). https://doi.org/10.1007/978-3-642-36046-6_1

46. Beyer, D., Stahlbauer, A.: BDD-based software verification: Applications to event-condition-action systems. Int. J. Softw. Tools Technol. Transfer **16**(5), 507–518 (2014). https://doi.org/10.1007/s10009-014-0334-1
47. Beyer, D., Wendler, P.: CPAchecker with sequential combination and strategy selection. In: Automatic Software Verification. Springer (2024)
48. Beyer, D., Wendler, P.: CPAchecker releases. Zenodo. https://doi.org/10.5281/zenodo.3816620
49. Beyer, D., Wendler, P.: CPAchecker release 3.0. Zenodo (2024). https://doi.org/10.5281/zenodo.12663059
50. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
51. Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., von Rhein, A., Apel, S., Beyer, D.: Facilitating reuse in multi-goal test-suite generation for software product lines. In: Proc. FASE. pp. 84–99. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_6
52. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
53. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
54. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
55. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Proc. SAS. pp. 215–237. LNCS 7935, Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_13
56. Friedberger, K.: CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis (competition contribution). In: Proc. TACAS. pp. 912–915. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_58
57. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Proc. ATVA. pp. 365–380. LNCS 8172, Springer (2013). https://doi.org/10.1007/978-3-319-02444-8_26
58. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
59. Jakobs, M.C.: CoVeriTest with dynamic partitioning of the iteration time limit (competition contribution). In: Proc. FASE. pp. 540–544. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_30
60. Jakobs, M.C.: CoVeriTest: Interleaving value and predicate analysis for test-case generation (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 847–851 (December 2021). https://doi.org/10.1007/s10009-020-00572-1
61. Jakobs, M.C., Richter, C.: CoVeriTest with adaptive time scheduling (competition contribution). In: Proc. FASE. pp. 358–362. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_18
62. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_15

63. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
64. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
65. Leeson, W., Dwyer, M.: Graves-CPA: A graph-attention verifier selector (competition contribution). In: Proc. TACAS (2). pp. 440–445. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28
66. Leike, J., Heizmann, M.: Ranking templates for linear loops. Logical Methods in Computer Science **11**(1) (2015). https://doi.org/10.2168/LMCS-11(1:16)2015
67. Löwe, S.: CPAchecker with explicit-value analysis based on CEGAR and interpolation (competition contribution). In: Proc. TACAS. pp. 610–612. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_44
68. Löwe, S., Mandrykin, M.U., Wendler, P.: CPAchecker with sequential combination of explicit-value analyses and predicate analyses (competition contribution). In: Proc. TACAS. pp. 392–394. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_27
69. Löwe, S., Wendler, P.: CPAchecker with adjustable predicate analysis (competition contribution). In: Proc. TACAS. pp. 528–530. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_40
70. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
71. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14
72. Peled, D.: Ten years of partial order reduction. In: Proc. CAV. pp. 17–28. Springer (1998). https://doi.org/10.1007/BFb0028727
73. Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
74. Ruland, S., Lochau, M., Jakobs, M.C.: HybridTiger: Hybrid model checking and domination-based partitioning for efficient multi-goal test-suite generation (competition contribution). In: Proc. FASE. pp. 520–524. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_26
75. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electr. Notes Theor. Comput. Sci. **149**(1), 79–96 (2006). https://doi.org/10.1016/j.entcs.2005.11.018
76. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. FMCAD, pp. 127–144. LNCS 1954, Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
77. The Open Group: 64-bit and data size neutrality. https://unix.org/whitepapers/64bit.html, accessed: 2024-06-29
78. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Proc. FMCAD. pp. 1–8. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351148
79. Vizel, Y., Grumberg, O., Shoham, S.: Intertwined forward-backward reachability analysis using interpolants. In: Proc. TACAS. pp. 308–323. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_22
80. Wendler, P.: CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proc. TACAS. pp. 613–615. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_45

81. Wonisch, D.: Block abstraction memoization for CPAchecker (competition contribution). In: Proc. TACAS. pp. 531–533. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_41
82. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Proc. ICFEM. pp. 332–347. LNCS 7635, Springer (2012). https://doi.org/10.1007/978-3-642-34281-3_24
83. Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., Novikov, E., Petrenko, A.K., Khoroshilov, A.V.: Configurable toolset for static verification of operating systems kernel modules. Programming and Comp. Softw. **41**(1), 49–64 (2015). https://doi.org/10.1134/S0361768815010065